Naval Research Laboratory

NRL/FR/5510--98-9876

The Hobbes Software Architecture for Virtual Environment Interface Development

Kapil Dandekar
James Templeman
Linda Sibert

Navy Center for Applied Research in Artificial Intelligence
Information Technology Division

Robert Page

ITT Systems & Sciences Corporation
Alexandria, Virginia

April 16, 1998

DTIC QUALITY INSPECTED 4

19980423 021

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (*Leave Blank*)	2. REPORT DATE April 16, 1998	3. REPORT TYPE AND DATES COVERED Final Report	
4. TITLE AND SUBTITLE The Hobbes Software Architecture for Virtual Environment Interface Development			**5. FUNDING NUMBERS** PN - 55-7029 PE - 0602233N
6. AUTHOR(S) Kapil Dandekar, James Templeman, Robert Page,* and Linda Sibert			TA - 03328
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Research Laboratory Washington, DC 20375-5320			**8. PERFORMING ORGANIZATION REPORT NUMBER** NRL/FR/5510--98-9876
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research 800 North Quincy Street Arlington, VA 22217-5660			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES *ITT Systems & Sciences Corporation Alexandria, VA 22303			
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			**12b. DISTRIBUTION CODE**

13. ABSTRACT (*Maximum 200 words*)

This report describes the design and use of the Hobbes system. This system is a framework for rapid interface development for high performance graphics and virtual reality (VR) applications. The design of the system incorporates the following fundamental features: object-oriented design, transparency of the underlying windowing system and graphics application program interface (API), transparency of multiprocessing and shared memory systems, and portability and extendibility. The system also supports the following capabilities: creation of real-time three-dimensional graphics applications; simultaneous, independent use of multiple I/O devices (not limited to mouse and keyboard); and transparent, high throughput use of local or networked I/O devices. The reader of the report will acquire an appreciation of the potential benefits of the Hobbes system as a tool for both development and research, particularly concerning applications that require rapid prototyping of novel human-computer interaction (HCI) techniques. The reader will also gain an understanding for how the Hobbes system architecture could evolve to include the functionality of a general virtual environment (VE) testbed. The current C++ implementation of the Hobbes system runs on Silicon Graphics workstations and makes extensive use of the Performer application development environment.

14. SUBJECT TERMS		15. NUMBER OF PAGES 38
Virtual reality Evaluation tool Application framework Interactive graphics		16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

CONTENTS

EXECUTIVE SUMMARY

This report describes the design and utilization of the Hobbes system. This system is a framework for rapid interface development for high-performance graphics and virtual reality (VR) applications. The design of the system incorporates the following fundamental features:

- object-oriented design,
- transparency of the underlying windowing system and graphics application program interface (API),
- transparency of multiprocessing and shared memory systems, and
- portability and extendibility.

The system also supports the following capabilities:

- creation of real-time three-dimensional (3-D) graphics applications;
- simultaneous, independent use of multiple I/O devices (not limited to mouse and keyboard); and
- transparent, high throughput use of local or networked I/O devices.

The reader will acquire an appreciation of the potential benefits of the Hobbes system as a tool for both development and research, particularly concerning applications that require rapid prototyping of novel human-computer interaction (HCI) techniques. The reader will also gain an understanding for how the Hobbes system architecture could evolve to include the functionality of a general virtual environment (VE) testbed. The current C++ implementation of the Hobbes system runs on Silicon Graphics workstations and makes extensive use of the Performer application development environment.

THE HOBBES SOFTWARE ARCHITECTURE FOR VIRTUAL ENVIRONMENT INTERFACE DEVELOPMENT

1. INTRODUCTION

The Hobbes system is an object-oriented framework for interface development for high-performance graphics and virtual reality (VR) applications. It is designed to facilitate the addition of prototype input and output devices for the purpose of rapidly evaluating new human-computer interaction (HCI) techniques. As part of this evaluation procedure, the Hobbes system supports the *instrumentation* of the HCI technique. Instrumentation refers to the ability of the HCI developer to view and record the actions of test subjects. The Hobbes system provides an object-oriented, modular, development facility to fulfill these needs while providing relative transparency to the underlying windowing system and graphics application program interface (API). Fundamentally, the system incorporates the following features:

- object-oriented design,
- transparency of the underlying windowing system and graphics API,
- transparency of multiprocessing and shared memory systems, and
- portability and extendibility.

The system also supports the following capabilities:

- creation of real-time three-dimensional (3-D) graphics applications;
- simultaneous, independent use of multiple I/O devices (not limited to mouse and keyboard); and
- transparent, high throughput use of local or networked I/O devices.

Hobbes was designed to solve the broad problem of efficiently developing computer programs to test HCI techniques. Three-dimensional graphics and VR applications need to have a high frame rate so that the user feels that these applications are truly interactive. They need to support commercial and custom-made I/O device hardware and thus need a mechanism for ensuring that this support can be quickly provided and modified. These kinds of systems also need to reuse existing code in an efficient manner so that the application programmers can spend their time working on testing HCI techniques rather than having to relearn intricate details of the underlying windowing and graphics API for each new project.

A software package such as Hobbes specifically designed to prototype and evaluate HCI techniques is useful to both researchers developing new 3-D interfaces and to human factors and experimental psychologists studying the way people use and react to VR. As a tool for both development and research, Hobbes supports the creation of both finished applications and instrumented applications.

The current version of the Hobbes system contains the following components:

- Object-oriented System Manager class hierarchy used to rapidly prototype high-performance graphics applications. This hierarchy allows these applications to be developed while providing transparency to the underlying windowing system, graphics API, and system multiprocessing capability.
- Object-oriented Device class hierarchy used to permit the development of new device drivers for local or networked I/O devices using a wide variety of communications protocols.
- Device drivers supporting several commercial and custom-made input devices.
- The Communications Manager, a separate process that allows for independent, high throughput use of local or networked I/O devices while providing transparency to an underlying shared memory information transfer paradigm.
- Object-oriented graphics class hierarchy used to provide a library of graphics primitives while providing transparency to the underlying graphics API.

2. BASIC SYSTEM CONCEPTS

The following sections describe the rationale behind the design of the Hobbes system. This design primarily addresses the concerns of the application programmer. Specifically, the system is designed to ensure that the application programmer does not have to "reinvent the wheel" with each new project. While, in principle, this is the modus operandi of all computer programmers, the design of the Hobbes system is such that low level graphics and window code is reused and project development is as productive as possible. Please refer to Section 4 for a discussion comparing Hobbes with some other existing systems.

Hobbes is not an independent, stand-alone system. It is a framework that utilizes the power of the fundamental systems that it is built upon. It does not seek to totally isolate the application programmer from these fundamental systems. Hobbes encapsulates the basic functionality of the underlying systems so that the application programmer does not *need* to know the specific details of the operation of these systems. The application programmer may still directly access the features of these fundamental systems. Thus, the Hobbes system is more of an unobtrusive bookkeeper that allows for uniform application design. The framework remains the same while the details of implementation for the specific project are left to the application programmer.

2.1   Real-Time Graphics API

The Hobbes system is currently built upon the IRIS Performer application development environment for creating real-time graphics applications on SGI workstations. IRIS Performer is the development environment of choice for the kinds of applications that Hobbes is intended to facilitate. Specifically, Performer supports many different types of database formats needed in the construction of virtual environments (VEs). It also contains routines for the efficient storage and rendering of these complex scenes while allowing the introduction of a transparent multiprocessing capability. It should be emphasized, however, that Hobbes does not seek to provide wrapper code for all the functionality possible in Performer. To do that would be both unnecessary and wasteful. Rather, the purpose of the Hobbes system is to provide easy, transparent access to Performer functionality (coupled with that of X-Windows, OpenGL, and that of I/O devices) while still allowing the availability of the full power of Performer for programmers who are proficient using Performer. In other words, choosing to use the Hobbes system does not require the application programmer to give up any of the functionality of Performer. However, choosing Hobbes allows novice Performer users to gain

access to a large amount of Performer functionality with a smaller initial learning curve and thus allows for the rapid prototyping of high-performance graphics applications.

2.2 Object-Oriented Design

One of the guiding principles in the design and utilization of the Hobbes system is object-oriented programming. The object-oriented language used in the initial implementation of the Hobbes system is C++. This language was chosen because it not only supports object-oriented programming, but also because it is highly compatible with IRIS Performer. Past development systems have an underlying object-oriented design, which is accessed and utilized in a nonobject-oriented fashion. These systems are object oriented only up to the level at which the application programmer uses the system. If the application programmer desires an object-oriented implementation, the transition often requires much time and effort.

The Hobbes system has an object-oriented design extending from the lowest levels of the system up through and including code written by the application programmer. This paradigm allows the functionality of Hobbes' underlying systems along with many initialization tasks to be encapsulated. If this low-level code needs to be modified, it is accessible; otherwise, this code is effectively "hidden" from the application programmers so that they can start project development at a higher level and minimize work time. This allows for a short project start-up time and allows the rapid prototyping of Hobbes applications.

As with any effectively designed C++ system should the need eventually arise, the Hobbes architecture could be maintained while using a different application development environment. For example, the system could be moved from X windows to Windows NT by changing the window and event-handling code. This would allow Hobbes application code to be relatively portable from one platform to another.

2.3 Application Structure

Multiple windows can be created in the Hobbes system. Each of these windows can have graphical objects displayed from multiple viewpoints. An application, in the terminology of the Hobbes system, is a logical grouping of windows so that each individual group performs some overall, well-defined task. The Hobbes system is capable of simultaneously running multiple applications simply through the addition of a line in the system configuration file (discussed in Appendix A). It may be easier to think of applications as "plug and play" modules, with each module capable of multiple windows on screen, that may perform their given purpose either in standalone operation or with other tasks simultaneously.

The following example illustrates two ways in which the same program may be implemented in the Hobbes system. One approach has the program written as a single application with multiple windows, whereas the other consists of a group of applications with a small number of windows each. Either of the approaches outlined in the paragraphs below could be used by the application programmer. However, the discussion is included to present both sides of the issue from the standpoint of the Hobbes system and illustrates how instrumentation can be implemented in a versatile way.

Consider a maneuvering task over a virtual landscape in which a custom-made input device is used. This task consists of several different windows. One window shows a graphical representation of the terrain map. Another window shows a schematic of the custom-made input device, while the final

window shows an X-Y plot of the data received from individual sensors on the custom-made input device. Conceivably, this could all be considered to be one application or task when broadly thinking of all of the windows grouped together as the "Maneuvering Task with the Custom-Made Device." However, there is another way of considering this problem made possible through the use of the Hobbes system that allows greater flexibility.

All of the windows mentioned above may not always be required. Considering the situation above, the input device schematic and X-Y plot can be thought of as instrumentation functions only. If the terrain map window was used to drive a head mounted display, the window would require as high a graphics throughput as possible. A conventional solution to this problem, such as that in the first method, would be to comment out the code for whatever group of windows that is not being used at any given moment and recompile the program. This has several disadvantages. First, the input device data in conventional programs are usually collected specifically for one of the windows with the information being passed along to the other two. The Hobbes system can let you bypass this difficulty by breaking the above simulation into two applications. In this method, each application in the Hobbes system has independent access to all of the input devices. Another disadvantage of the first method's conventional solution is that it requires recompiling. When pilot studies are being run while further application development is still going on, this can be inconvenient and wasteful. Setting up the above scenario as two applications—one with the terrain overview (single window application) and one with the instrumentation windows—solves these problems. The Hobbes configuration file (discussed in Appendix A) allows these subsystems to run individually or together easily without having to recompile. Furthermore, these individual instrumentation applications are highly portable from one task to another.

2.4   Event Handling

The Hobbes system supports a structured system for event handling that relies on the specification of application-programmer-defined callbacks. Application programmers can "register" event handlers with the applications that they write. These event handlers can be specified for high-level events sent to applications, or for low-level events sent to individual windows. By design, these programmer-defined event handlers are given access to the public interface of the application/window in which the event occurred.

In the current version of the Hobbes system, the X Window System is used to provide a mechanism for event handling and low-level window management. Performer supports the use of X event-handling and its own internal event-handling system. Hobbes uses XEvents nearly exclusively because it allows greater flexibility in communicating between any kind of window, Performer or otherwise. For example, inter-application communication is facilitated through the use of X client message events. In addition, the handshaking needed for the transfer of data between applications and I/O devices is facilitated through X client message events.

In the future, a different windowing system could be supported using the Hobbes architecture. All X-specific routines required by the system are localized in the C++ code hierarchy and could be replaced with routines specific to another event handling system (e.g. Windows NT).

2.5   I/O Devices

A major design feature of the Hobbes system is to provide independent, easy access to input devices. Input devices are not limited to just the mouse or the keyboard. The Hobbes system contains a C++ device hierarchy used to develop interface classes for novel input devices. These input device

classes can then be used to produce Hobbes device drivers, which provide data to any number of Hobbes applications requesting data from them. Using the Hobbes system can provide a uniform interface to any number of custom-made I/O devices, which will greatly decrease the amount of time it takes to develop novel hardware with new applications.

Hobbes device drivers are separately running processes whose sole function is to transfer data to and from a given I/O device. Device driver processes transfer data to or from a given location in shared memory where any number of Hobbes applications can readily and independently access it. The handshaking required to initiate this shared memory connection is facilitated through the use of X client message events.

3. ARCHITECTURE COMPONENTS

The running system operates on multiple levels in several processes. On the top level, in the main process, is the **System Manager**, which handles applications, windows, a graphical user interface (GUI), and the relaying of events. Below the System Manager are **Applications** which can contain any number of **Windows**. Each window can similarly contain any number of **Views**. Processing time for both applications and views is handled through the use of callback routines executed by the System Manager either sequentially or through the use of the system multiprocessing capability. Figure 1 illustrates this structure.
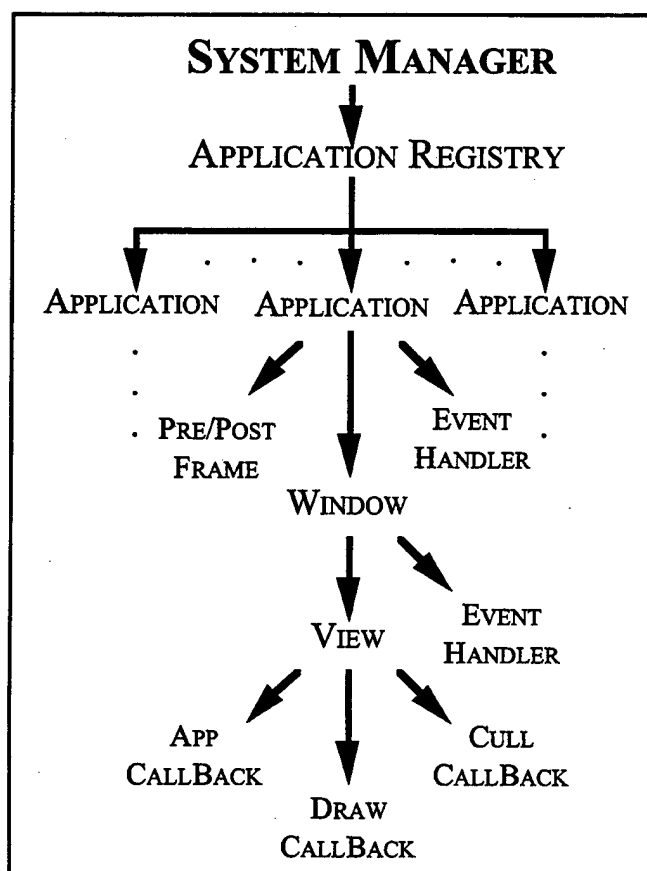

Fig. 1 – System Manager hierarchy

In a separate process, the **Communications Manager** sets up the interaction through shared memory between any number of individual input device processes and the applications managed by the System Manager. The structure of the Communications Manager (illustrated in Fig. 2) will also be discussed in greater detail later.
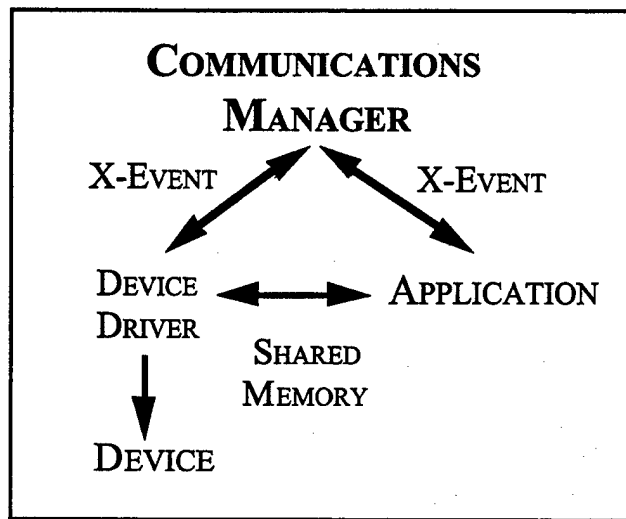

Fig. 2 – Communications Manager

3.1   System Manager

The System Manager is the heart of the Hobbes system. It handles all basic system and Performer initialization, oversees the storage and dynamic allocation of applications, and contains routines for distributing XEvents to applications and windows as needed. The System Manager also contains the main program loop that allocates time for individual application processing, event handling, and graphics rendering.

In the main loop of the System Manager process, the first step is to sequentially allocate pre-frame rendering time to individual applications for application-specific processing. Next, channel-specific processing and rendering is handled through the use of multiprocessing. After that, individual applications are allocated post-frame rendering time for further application processing. Finally, the graphical user interface is updated and events are pulled off the event queue of the underlying window manager (e.g., X) and distributed to application-specific callback routines. Each application-specific callback routine may further choose to distribute the event along to individual windows contained within the application.

Note that the System Manager does not deal directly with individual applications. The *Application Registry* class parses application configuration files and oversees the actual allocation of Performer resources to any given application. The Application Registry is a class and an object of this class is contained in the System Manager.

The application programmer should have very little need to directly modify or use the System Manager. A notable exception to this rule is that the System Manager may be directly invoked to query the overall system time or any other system parameter contained within the *System Manager*

class. Another exception to this rule involves the use of the routine that informs the System Manager that an application has signaled for system termination. This routine is most often used in event-handler code.

3.1.1 Applications

The *Application* class is an abstract base class from which specific application classes are derived. The Application class has basic routines defined for the creation and management of windows and viewports, event handling, and pre- and post-frame routines that are accessed by the Application Registry should the application be registered. It also has member functions that define the condition through which the application may be exited and the function to execute when the application exits.

Since the Application has pure virtual functions, an object of type *Application* cannot be declared. Derived applications should contain specific data and methods relevant to each particular application. With this approach and with the nature of the supplied callback routines, there should be no need for global variables. This discussion is not meant to imply that there can only be a single level of inheritance by all application programmer defined applications. For large systems, application programmers can use object-oriented programming to develop a hierarchy of objects derived from the Application abstract base class.

Individual windows inside of an application all share the same application data and can communicate with each other in a straightforward manner. Applications are enscapulated; they do not share data with one another directly but rather rely on event sending through the System Manager to share information and communicate. This modular "plug and play" aspect allows for greater run-time flexibility and ease of development.

For a detailed explanation of the implementation of the Application class in the current version of the Hobbes system, refer to Appendix B.

3.1.2 Windows

The *Window* class is responsible for storing all information regarding a single window on the screen. In the current implementation of Hobbes, the main component of this class is the Performer pfPipeWindow. While the Hobbes system design philosophy seeks to hide the details of implementation to allow a quick start-up time, the full functionality of the Performer window object can be accessed by the application programmer if needed. In a completely analogous manner, the functionality of X-Windows can be accessed through the Performer window object. Methods and data also exist to perform event handling (assuming the specific application in question passes events down to the window level), coordinate with X, and handle as many Views as the application programmer specifies.

3.1.3 Views

In the terminology of Performer, a *View* is essentially a pfChannel that includes scene information. Specifically, information about the viewing volume of the scene is stored along with low-level callback routines for drawing and culling. This class has some very specific default actions to perform when setting up a View, especially when all that is specified in the View initialization is model file name. In this case, a default viewing volume is set up that can easily be modified through manipulation of the Performer pfChannel object inside of this class.

3.1.4 Graphical User Interface

Every program developed in the Hobbes system has a single GUI associated with it. This GUI can have widgets (buttons, sliders, dials, etc.) added to it by each application managed by the System Manager. The System GUI informs the owner application of widget activation through the use of X client messages. Thus, any application that adds a widget to the GUI should define a client message handler for itself to handle widget activation/manipulation.

The current version of Hobbes only supports a primitive method for adding widgets to the System GUI. The application should define the position and size of its GUI widgets relative to a current reference point stored in the System GUI. Thus, the System GUI does not explicitly enforce the fact that widgets should not overlap. Future releases of Hobbes may provide some form of GUI/Application editor.

3.2 Communications Manager

The Communications Manager is the process responsible for registering applications and devices and transmitting data between applications and device driver processes. In the registration process for an input device, applications specify the type of data to be collected while device driver processes specify the type of data provided. The Communications Manager then handles the initial matching of applications seeking a particular type of data with the input device process providing the given data type. The transfer of data between application and device driver is facilitated through the use of UNIX System V shared memory. The Communications Manager continues to mediate the relationship between application and device driver during the execution of the program to ensure that either the application, in the System Manager process, or device driver process knows of any change in status (e.g., exit, crash, etc.) of the other. Through the use of this approach, all applications in the System Manager can have robust, simultaneous, and independent access to as many input devices as is required.

3.3 I/O Devices

There are two relevant constructs to examine when considering the operation of input/output devices. The first is the *Device*, a class containing the methods and data necessary to initialize and interact with the I/O hardware. The second is the *Device Driver*, an individual process that uses a specific Device object to read data from an input device and store the data in shared memory so that applications may access it. A device driver for an output device would take data stored in shared memory by applications and use the Device class to transfer the data. It is important to realize that this is not a UNIX kernel level driver communicating directly with hardware but, rather, a user process level driver.

Hobbes device drivers make use of existing low-level drivers (serial connection, socket, etc.). For the convenience of the application programmer, Hobbes provides a library of these UNIX level drivers in a C++ class hierarchy (illustrated in Fig. 3). This hierarchy provides support for many different kinds of communications links in a uniform manner. This means that there is a uniform manner in which the application programmer accesses all devices, with commands such as "open," "close," "read," and "write." The details of the implementation of each particular kind of communications link is effectively hidden from the application programmer through this C++ implementation. Application programmers can implement a particular kind of device by deriving their Device class from the appropriate communications link in the hierarchy.
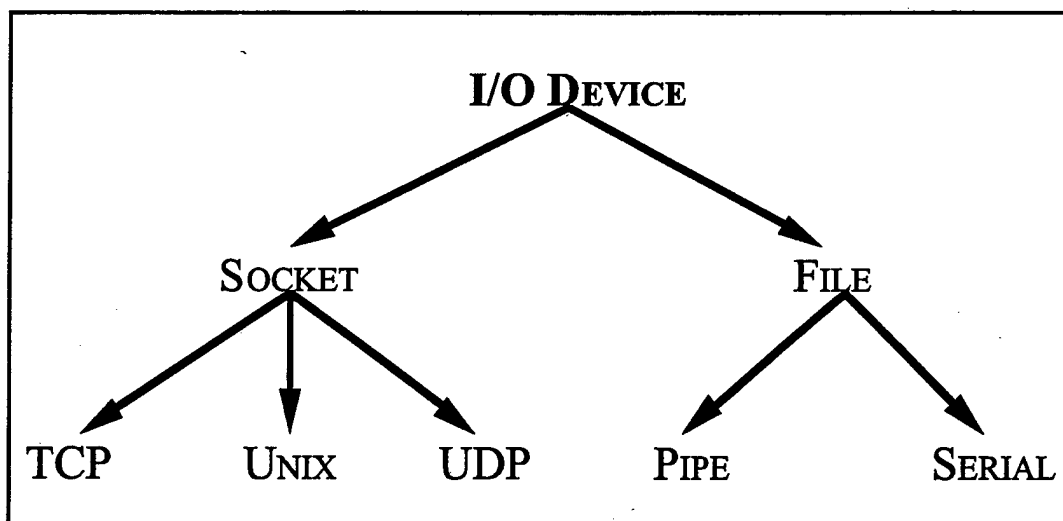
Fig. 3 – I/O device hierarchy

4. RELATED SYSTEMS

The Hobbes system is specifically designed to facilitate the implementation, refinement, and evaluation of new interaction techniques for use in VEs. This makes it hard to compare the current implementation of Hobbes with existing VE development systems. Most VE development systems emphasize tools for constructing animation-rich, complex VEs while maintaining a high level of graphics performance. However, the object-oriented framework of the Hobbes system was designed to grow to include the functionality of conventional VE development systems. This extension is currently being implemented.

There are several factors to consider when comparing Hobbes to commercially available packages. The first major issue is that commercial packages are proprietary and cannot be readily modified to meet the needs of the application programmer. They may not be flexible enough to allow the addition of commercial and custom I/O devices and the smooth integration of novel HCI techniques. They also may not allow the addition of data gathering and visualization tools (instrumentation) at different levels throughout the code. The easily accessible object-oriented structure of the Hobbes system, along with the included class libraries, were designed to encourage and support these kinds of modifications. Hobbes supports the development process and provides a framework for complete VE systems.

Related existing systems fall into two main categories: object-oriented graphics systems and complete VE development systems. These systems are discussed in greater detail in the following sections.

4.1 Object-Oriented Graphics Systems

Object-oriented graphics systems deal with graphics structures in much the same manner as the Hobbes system. Specifically, they make the transition from traditional methods of graphics to complete object-oriented programming implementations. For example, the *GRaphics using Object-*

Oriented Programming (GROOP) system, developed by Koved and Wooten [6], follows the same end-to-end object-oriented design methodology that is present in the Hobbes system. The GROOP system is designed to help programmers who know object-oriented programming but may not necessarily know about graphics programming. This idea of helping programmers through the use of object-oriented design is also a central theme in the Hobbes system. The current implementation of the Hobbes system contains a similar kind of object-oriented graphics hierarchy meant to aid programmers by providing a level of abstraction to the functionality of OpenGL.

For the Hobbes system to become a complete VE system, the object-oriented graphics hierarchy needs to be further developed. This involves the addition of an object-oriented system that enscapulates the geometrical and behavioral properties of objects in VR. A system that is designed to fulfill this goal is the *Object Modeling Language* (OML) developed by Green and Halliday [5]. OML is a procedural programming language built on the *MR Toolkit* [8], discussed later, that is graphics-language independent and allows for 3-D object geometry and behavior specification. A similar system, developed at Brown University [9], goes one step further and integrates modeling, animation, and rendering into a single framework. While in concept, the Hobbes system could use modeling languages such as these, the finished language would have to fit the end-to-end object-oriented design paradigm of the Hobbes system.

4.2 Virtual Environment Systems

Another class of system that the Hobbes design philosophy can be compared to is that of complete VE systems. One system developed at Xerox Palo Alto Research Center [7] relies on a system called the *Cognitive Coprocessor*. This Cognitive Coprocessor is a user interface architecture meant to support multiple, asynchronous data providers while providing for smooth animation. The concepts used by this system are present in many other VE systems. The design of the Communications Manager in the Hobbes system is analogous to this Cognitive Coprocessor architecture.

The *VB2* architecture, developed by Gobbetti and Balaguer [4], is similar to Hobbes because they are both designed for the rapid prototyping and testing of novel interaction techniques. They both also provide a basis for the construction of applications. VB2, built in the Eiffel object-oriented programming language, operates on a Decoupled Simulation Model that has device, application, and animation functions in separate processes communicating with standard inter-process communication (IPC) methods. This multiprocessing capability is similar to the combination of the Communications Manager and System Manager multiprocessing structure of the Hobbes system.

Another system is the MR Toolkit [8], developed at the University of Alberta. This library of subroutines is portable and has many applications including the building of VEs. It has support for a wide range of input and output devices and provides a C, C++, and Fortran 77 API. Unlike the Hobbes system, MR Toolkit does not emphasize rapid prototyping [3] nor does it contain an end-to-end object-oriented design. However, the MR Toolkit does support distributed computing. While the Hobbes system does not explicitly support distributed computing, the use of the Communications Manager, using distributed processors abstracted as input devices, can provide some of the same functionality. A more formal implementation of distributed computing in the Hobbes system is necessary for it to become a complete VE development system.

The *Virtual Environment Operating Shell* (VEOS) system, developed by Bricken and Coco [2], is another distributed VE construction system. In this system, unlike Hobbes, it is possible to replicate procedural or object-oriented programming techniques. This system also has a virtual entity

management system like that described in Section 4.1. However, this system is no longer supported [3].

The I/O device management paradigm in the Hobbes system is very similar in concept to those of existing VE development systems. Hobbes, however, is fairly unique when it comes to the strong end-to-end object-oriented design of the system. This object-oriented design extends from the lowest levels of the system up through and including the code developed by the application programmer. These concepts can be continued in Hobbes with the addition of a virtual entity management system and support for distributed computing.

5.   SHADWELL VIRTUAL ENVIRONMENT WALK-THROUGH EXAMPLE

The preceding sections have described the design philosophy and structure of the Hobbes system. In this section, a specific example is given to show how the Hobbes system could be used to develop a VE walk-through system. This first major application of the Hobbes system was the redesign of the Ex-USS *Shadwell* simulation initially developed by the Advanced Information Technology Branch, Code 5580, and the Navy Technology Center for Safety and Survivability, Code 6180, at the Naval Research Laboratory (NRL); and LCDR Tony King of the Naval Computer and Telecommunications Station.

This Ex-USS *Shadwell* simulation was originally developed to study shipboard damage control. The simulation is now being used by the Interface Design and Evaluation Section, Code 5513, to study locomotion techniques for VE walk-throughs. Novel input devices are being designed and constructed as part of this research project. The overall purpose of the study is to compare three different techniques for maneuvering in a VE. The methods compared are travel where looking (head directed), travel where pointing (hand directed), and travel through gestural walking.

The original Shadwell code was procedural and was developed for a very specific purpose. The need for more flexibility in this code was clearly identified. The Hobbes system is well suited to the requirements of this project for the following reasons:

- The C++ structure of the Hobbes system and applications allow greater flexibility and extendibility for future development.
- The multi-application and instrumentation capability, along with the I/O device hierarchy of the Hobbes system, allows for rapid input device development and iterative refinement.
- The "plug and play" nature of input device drivers allows for a rapid insertion of different device controllers and HCI techniques.
- Minimal knowledge of X and Performer will be required for future development because many X and Performer details are handled transparently by the system.

Since this project is fairly complex, code will not be provided to illustrate how to develop this specific application in the Hobbes system. However, Appendix C gives a comprehensive example of writing a simple Hobbes graphics application.

5.1   Application Development

The first step in the redesign of the Shadwell simulation under the Hobbes system was to consider the general structure of the necessary set of Hobbes applications. A novice programmer might be inclined to derive a "Shadwell_Application" class off the Application abstract base class that contains

everything. However, this solution does not make effective use of the Hobbes design paradigm, nor does it consider the critical issue of code reuse.

A more effective solution is to have multiple Hobbes applications. Some of these applications are derived directly off the Application abstract base class for instrumentation while others drive the primary display. For example, graphics windows were created for instrumentation applications such as graphs of sensor output values and pictorial representations of various novel input device operation These instrumentation applications were used as diagnostics   during the development of the novel HCI techniques. They were also used as modular, supplemental illustrations of the final system. The structure of Hobbes allow these various components to be easily configurable and removable in the completed system.

Rather than have a single main Shadwell walk-through application derived directly off the Application abstract base class, several levels of inheritance were used. Anticipating the need for future VE walk-throughs, a *Virtual Walk-through* class was derived from the Application abstract base class. This Virtual Walk-through class contained general viewing routines for navigating through a generic model and was implemented as an abstract base class with virtual functions for the I/O subsystems. A more task specific application for the Shadwell Virtual Walk-through was derived from this class. This Shadwell Virtual Walk-through class implemented the elements of a VE walk-through specific to the model and HCI techniques used in the study. This general approach allows a large amount of code reuse making future VE walk-through projects easier. This aspect of the Hobbes design philosophy can be expanded upon in future releases to turn the system into a more well-rounded VE development system.

5.2   Novel I/O Devices

The application programmer working on the Shadwell VE Walk-through project needs to be able to rapidly prototype novel input devices. These devices consist of pressure sensors for gestural walking recognition, position sensors for avatar update information, and button information from a hand controller. Furthermore, these input device systems are continually being developed and refined so code flexibility is important. To develop these systems, the application programmer first needs to consider the basic mechanism through which data will be transferred (serial connection, sockets, tcp/ip, etc.). From this, the programmer uses the Hobbes I/O device hierarchy to develop a Hobbes device definition. Specifically, the application programmer derives the *Device Definition* class from the applicable class in the I/O device hierarchy. Using the Device class, the programmer then wrote a Hobbes Device Driver. Recall that this is a separate process that collects data using the Device class and places this data into shared memory. This device driver used code from a Hobbes library to interact with the Communications Manager.

The Hobbes design philosophy allows several convenient options to be added into the system. These conveniences are available because of the method in which input device drivers and applications register with the Communications Manager. Recall that when a device driver registers with the Communications Manager, the type of data available is included with the registration information. Thus, the device drivers for each of the three techniques that used to control maneuvering are written to supply information of the same type. Which technique is used is then determined by which device driver the user invokes. From the application's perspective, the change is completely transparent and the true "plug and play" nature of Hobbes I/O devices is realized. Alternatively, the device drivers could have been written to register with the Communications Manager using different data types. This would have allowed the choice of which technique to use to be determined at the application level. The first option is preferable in the context of testing different

HCI techniques because it allows for rapid switching between techniques without recompilation during an experiment.

6.   FUTURE ADDITION

The future direction of Hobbes involves extending the system into a complete VE development testbed. Several readily apparent additions can be made to the system in order to facilitate this extension. The first is the addition of an object-modeling language such as the one developed by Green and Halliday. Ideally, the object modeling language in the Hobbes system would provide a mechanism through which the 3-D object geometry of a VE could be specified. Furthermore, this extension would also include a facility for the specification of object behavior and characteristics both in terms of animation and interactive response.

Another planned extension to the Hobbes system is the addition of a *Human* class. This class would enscapulate the functionality of translating raw user input into virtual motion based upon a library of possible movement models. In addition, it would handle matters such as view updates, collision detection, and interaction with the environment. Furthermore, the class would provide a mechanism through which avatars could be added. A possible implementation of this class in the Hobbes architecture would be to represent the body of the human as a collection of specialized nodes. Each node would have a different function such as a "Head" node, which is tied to a particular Hobbes view, or a "Collision" node, which determines the bounding volume of the human for collision detection purposes. The Human class itself would be an abstract base class; descendent classes would define the particular geometry, motion model, and general characteristics of a particular kind of human. These specialized human classes would be used to provide a "wing man" view of a VE or define a realistic human avatar for interaction in the VE.

Another extension that would enhance the usefulness of the Hobbes system as a VE development testbed is support for distributed simulations. The Communications Manager concept in the current Hobbes system makes the addition of this support fairly transparent to the overall system architecture. This facility would allow multiple virtual avatars from remote machines to interact in a single VE. Such a system would be realized by having an output device driver of a running Hobbes system transmit data over a high-speed network to an input device driver of another running Hobbes system on a remote machine. This would allow several avatars to interact with one another in the VE.

7.   CONCLUSIONS

The Hobbes system provides many benefits for application programmers. The system provides a low start-up time to programmers not familiar with the underlying graphics and windowing API. From the lowest levels of the system all the way through and including the code developed by application programmers, Hobbes is an object-oriented system. For this reason, the system encourages strong object-oriented design along with code reuse from a development standpoint. The object-oriented design could also allow the underlying graphics and windowing API to be changed while retaining the modular system structure and application code.

Hobbes also provides a uniform interface to many different kinds of I/O devices. This interface along with the information transfer paradigm of the system allows simultaneous, independent, and high throughput use of local or networked devices. These features of the Hobbes system make it ideal for both rapid prototyping of different HCI techniques and the instrumentation of high-performance

graphics and VR systems. Furthermore, as discussed in Section 6, many possibilities exist in the current Hobbes architecture for expanding the system into a general VE testbed. The Hobbes system is a valuable tool for both research and development.

8. ACKNOWLEDGMENTS

9. REFERENCES

1.   P. Appino, J.B. Lewis, L. Koved, D. Ling, D. Rabenhorst, and C. Codella, "An Architecture for Virtual Worlds," *Presence* 1(1), 1-17 (1992).

2.   W. Bricken and G. Coco, "The VEOS Project," *Presence* 3(2), 111-129 (1994).

3.   Naval Air Warfare Center (Orlando) Report 96-002,"Software Design of a Virtual Environment Training Technology Testbed and Virtual Electronic Systems Trainer," S.W. Davidson (1996).

4.   E. Gobbetti and J. Balaguer, "VB2 An Architecture for Interaction in Synthetic Worlds," Proceedings of the ACM Symposium on User Interface Software and Technology, November 3-5, 1993, Atlanta, GA, pp. 167-178.

5.   M. Green and S. Halliday, "A Geometric Modeling and Animation System for Virtual Reality," *Communications of the ACM* 39(5), 46-53 (1996).

6.   IBM Research Division Report RC 18732, "GROOP: An Object-Oriented Toolkit for Animated 3-D Graphics," L. Koved and W. Wooten, 1993.

7.   G. Robertson, S. Card, and J. Mackinlay, "The Cognitive Coprocessor Architecture for Interactive User Interfaces," Proceedings of the ACM Symposium on User Interface Software and Technology, November 13-15, 1989, Williamsburg, VA, pp. 10-18.

8.   C. Shaw, M. Green, M. Liang, and Yunqi Sun, "Decoupled Simulation in Virtual Reality with the MR Toolkit," *ACM Transactions on Information Systems* 11(3), 287-317 (1993).

9.   R. Zeleznik, D. Conner, M. Wloka, D. Aliaga, N. Huang, P. Hubbard, B. Knep, H. Kaufman, J. Hughes, J., and A. van Dam, "An Object-Oriented Framework for the Integration of Interactive Animation Techniques," *Computer Graphics* 25(4), 105-112 (1991).

Appendix A
SYSTEM CONFIGURATION FILE

A system configuration file is used in the Hobbes system to specify various system parameters and determine what applications should be loaded at run-time by the system. The default system configuration file is the .hobbesrc file. However, to specify a new configuration file, simply specify the name of that file in the command line of the hbLoader command. For the most part, all values specified in the system parameter section have default values that can be overridden in the system configuration file. This file is processed by the C++ preprocessor so C++ include files and commenting may be used. Here is what a sample file might look like:

```
// Hobbes System Default Configuration File
#include "Performer/pf.h"
#include "hbAppsList.h"

#define TRUE 1
#define FALSE 0

beginConfig
beginSystem
    numPipes      1                    // Number of pipes in the system
    useSync       TRUE                 // Indicates whether or not synchonization
                                       // should be used to attain given frame rate.
    phaseMode     PFPHASE_LOCK         // Frame rate management parameters
    frameRate     60.0
    videoRate     60.0
    guiOrigin     0 0                  // Graphical User Interface parameters
    guiSize       400 400
    beginFilePath                          // File path in which to search for model
data
        ./textures
        /usr/share/Performer/data
        /usr/demos/models
        /usr/demos/data/flt
    endFilePath
endSystem

beginAppList
    App SPACESHIP Space1 spaceShip.hb
    App EARTHSKY Esky1 earthSky.hb
endAppList
endConfig
```

System specific values are given in the begin/end System section of the file. In the begin/end AppList section, application programmer defined applications may be specified. Applications that may be thought of as modules may each be run independently or with other modules at the application programmer's discretion. In the above example, two different applications are invoked. The App command tells the file parser that a new application is to be added. The next parameter is a unique constant associated with the application defined in hbAppsList.h. The following string is a unique name associated with that particular invocation of the application, since multiple applications of the same type can be started simultaneously, that name is used to distinguish them. Finally, the last parameter is the application configuration file name. The method in which this file is parsed is left completely up to the application programmer in the application member function parseFile. If no application configuration file is required, simply provide a dummy file name and leave the application's parseFile member function blank.

APPLICATION CLASS IMPLEMENTATION

The Application abstract base class is defined like this:

```
// Class definition
class hbApplication
{
 public:
     // Constructors
     hbApplication( );
     hbApplication( char *newName, char *appFile );

     // Destructors
     ~hbApplication( );

     ...

     // Trigger application exit
     void setExitFlag( );
     // Check for application exit
     int getExitFlag( );

     ...

     // Pure virtual functions that should be defined by derived classes
     virtual void startUp( ) = 0;        // Start up for each application
     virtual void parseFile( ) = 0;      // Parse application datafile
     virtual int  exitCondFunc( ) = 0;
                  // Function to check for application exit
     virtual void exitFunc( ) = 0;       // Execute upon application exit
     virtual void preFrame( ) = 0;
                  // Pre-frame routine (access by Application registry)
     virtual void postFrame( ) = 0;
                  // Post-frame routine (access by Application registry)

     // Window handling routines
     // Allocate a window for the application
     int allocateWindow( char *WinName );

     // Initialize a window in the application
     int initWindow( char *WinName, int whichPipe,
                  int originX, int originY, int sizeX, int sizeY );
```

```
    int initWindow( char *WinName, pfPipeWindow *addWindow );


    // View handling routines
    ...

    // Event handling routines
    void addEventHandler( int event_type,
                 void (*ptr)( hbEventHandler &event, void *data ) );
    int  removeEventHandler( int event_type );
    ...



    // Communications manager routines
    void allocateDevice( int dataType, hbAppWindow *devWindow,
                            size_t bufferSize );
    int initDevice( int dataType, key_t deviceKey );
    void *getData( int dataType );
    ...

 private:
        ... Internal data ...
};
```

Several notable observations can be drawn through examination of this Application base class. The first is that this class is an abstract base class, that is, it contains pure virtual functions. The existence of these functions alone implies that objects of this type cannot be declared. Only objects based upon descendants of this type may be declared if these pure virtual functions are defined as regular C++ member functions.

From the constructor of this class, it should be noticed that all applications have a name and file name associated with them. If the constructor without parameters is for some reason used, a default name and file name will be assigned to the application. Application constructors are called by the Application Registry in the System Manager. Please note that there is a distinction between what initialization code should go in the application constructor and what code should go in the startUp subroutine. The need for this distinction is dictated by the general structure of Performer programs. All Performer programs start off with initialization of objects that will be placed into shared memory that will subsequently be visible to all other processes in the program. These multiple processes are created at the invocation of the pfConfig command (note: the "pf" prefix indicates a Performer routine). After this point, data may not be added to shared memory, but graphical objects may be initialized and opened for later processing in the main loop of the program. In other words, the space for all Hobbes objects must be allocated in the constructor of the objects and executed before the pfConfig. The actual act of initialization of these objects must occur in the startUp member function.

The application name should be a distinct name given to each application. The application file name may be used for application-specific configuration information. The parsing of this file should be handled in the parseFile member function in the application. The actual format of the file and the method of parsing is left up to the application programmer. It should be noted that should an application

configuration file not be required, a dummy file name can be provided and the `parseFile` member function can be left blank.

Also note that each application has an exit flag. When the constructor for an application is called, the exit flag is initialized to zero. At any point inside of the code of the application, this exit flag can be set. In this event, the System Manager will know that the application is ready to terminate execution. Prior to removing this application from its application list in the Application Registry, the `exitFunc` of the application is invoked. This is user-defined code, which needs to be written for each application. In this routine, various shutdown tasks should be called for the application followed by the command: `delete this`. This is required to free the memory for the application and call the application destructor. An exit flag may also be set for the System Manager in this `exitFunc` routine to signify that the entire system should be terminated with the termination of the application in question. If this flag is not specifically set in the System Manager, the application with the set exit flag will be stopped, while all other applications managed by the System Manager will continue operation.

A member function related to the discussion of terminating applications is the `exitCondFunc` member function. This function is called every time that the application is given time to execute. Based upon the criteria placed within it by the application programmer, if the function returns zero, application operation will continue unhindered. However, if the function returns a non-zero value, the exit flag for the application will be set and application operation will be terminated through the actions mentioned in the preceding paragraph.

The `preFrame` and `postFrame` member functions refer to code executed prior to and after the rendering of the current frame. This code is called for each application at the appropriate time in the main loop of the System Manager, using the application list stored in the Application Registry.

The use of all of these member functions would be best demonstrated through the use of a specific example that is given in Appendix C.

Once the class is written, it must be integrated with the existing code. This is done in three steps:

1. Edit the file `hbAppsList.h` and define a name to be associated with a given application with a unique integer value. For example, a sample `hbAppsList.h` file might look like this:

```
// hbAppsList.h - Application List header file
// Kapil Dandekar
// dandekar@itd.nrl.navy.mil
// Interface Design and Evaluation - Code 5513
// Navy Center for Applied Research in Artificial Intelligence
// United States Naval Research Laboratories

#define SPACESHIP 1
#define EARTHSKY 2
```

In the example shown above, for a new application class of type `MyApplication`, the following line could be added to the file.

```
#define MYAPPLICATION 3
```

2. Edit the file `hbAppReg.C` and add an `#include` directive to include the header file containing your new application header file. Also, edit the function void `hbApplicationRegistry::startApplication` and add a case statement to invoke your new application. For example, `hbAppReg.C` might look something like this before editing:

```
// hbAppReg.C - Application Registry class definition file
// Kapil Dandekar
// dandekar@itd.nrl.navy.mil
// Interface Design and Evaluation - Code 5513
// Navy Center for Applied Research in Artificial Intelligence
// United States Naval Research Laboratories

// General Include files
#include <stdio.h>

// Hobbes Include files
#include "hbAppReg.h"
#include "hbAppsList.h"

// Application Include files
#include "spaceShipapp.h"
#include "earthSkyapp.h

void hbApplicationRegistry::startApplication( int appCode,
                            char *appName, char *appFile )
{
        hbApplication *myApp;
        switch(appCode)
        {
                case SPACESHIP:
                        myApp = new SpaceShipApp( appName, appFile );
                        myApp->parseFile( );
                        addApplication( myApp );
                break;

                case EARTHSKY:
                        myApp = new EarthSkyApp( appName, appFile );
                        myApp->parseFile( );
                        addApplication( myApp );
                break;

                default:
                        cout << "Application code "
                                << appCode << " undefined!" << endl;
                break;
        }
}
```

For the example mentioned above, add the line #include "MyApplication.h" in the Application Include Files section. In the `startApplication` member function, add the lines:

```
case MYAPPLICATION:
        myApp = new MyApplication( appName, appFile );
        myApp->parseFile( );
        addApplication( myApp );
break;
```

3. Finally, if you desire to load the application at program run-time, edit the System Configuration File and add the appropriate line to the application list. For example, if the file looks like this:

```
// Hobbes System Default Configuration File
#include "Performer/pf.h"

#include "hbAppsList.h"

#define TRUE 1
#define FALSE 0

beginConfig

beginSystem
        numPipes        1
        useSync         TRUE
        phaseMode       PFPHASE_LOCK
        frameRate       60.0
        videoRate       60.0
        guiOrigin       0 0
        guiSize         400 400
        beginFilePath
                ./textures
                /usr/share/Performer/data
                /usr/demos/models
                /usr/demos/data/flt
        endFilePath

endSystem

beginAppList
        App SPACESHIP Space1 spaceShip.hb
endAppList
```

If the new application is to replace the application currently listed, comment out or delete the current entry in the begin/end `AppList` section and add a line like the following:

```
App MYAPPLICATION TestApp test.hb
```

The "App" in that line tells the file parser that a new application is being added. The MYAPPLICATION is the flag defined in hbAppList.h and is used in hbAppReg.C. The TestApp is a name given to that particular invocation of application MyApplication (multiple invocations with unique application names are possible). The test.hb is a configuration file for the application that is parsed by the parseFile member function in the application class. If no application configuration file is required, simply provide a dummy file name and leave the parseFile member function empty.

Appendix C
HOBBES APPLICATION EXAMPLE

Now we describe in detail how a simple application using the Hobbes system can be written. This application draws two spaceships in a single window with each in their own view. This window has event handlers defined to handle keyboard input and mouse button presses. The user input simply changes the scaling of the animated spaceships in the window. These event handlers are also used by the application to interface with the System Manager GUI and with the Communications Manager. The application also has another window that can be used to provide another place where mouse input can be applied to the spaceship window. The window with the spaceships has application callbacks defined to allow the ships to rotate.

This small application is meant to illustrate how the Hobbes system can be used to quickly prototype a simple graphics application that relies on mouse input. Mouse input was chosen for illustration purposes since it is universally available on today's graphics workstations. Using traditional techniques, the development of this simple application would require a large amount of time for a beginner programmer because details of the underlying windowing and graphics API would have to be handled. Hobbes alleviates this large start-up cost and allows study of the HCI technique to take place as soon as possible.

The header file for our sample application looks like this:

```
// Space Ship application code header file
// Kapil Dandekar

#ifndef SPACESHIP_APP_H
#define SPACESHIP_APP_H

#include "hbApplication.h"
#include "hbFastrakData.h"
#include "hbBirdData.h"

class SpaceShipApp : public hbApplication
{
    public:
      SpaceShipApp( );
      SpaceShipApp( char *newName, char *appFile );
      ~SpaceShipApp( );

      // Define pure - virtual functions from abstract base class
      void startUp( );
      void parseFile( );
      int exitCondFunc( );
```

```
        void exitFunc( );
        void preFrame( );
        void postFrame( );

        // Application specific routines
        pfuWidget *getHideWidget( ) { return( HideWidget ); }
        int getHideToken( ) { return( HideToken ); }

        float getEntExtentRadius( ) { return(Ent_extentRadius); }
        void setEntExtentRadius( float newVal )
                { Ent_extentRadius = newVal; }

        float getKlgExtentRadius( ) { return(Klg_extentRadius); }
        void setKlgExtentRadius( float newVal )
                { Klg_extentRadius = newVal; }

    private:
        float Ent_extentRadius;
        float Klg_extentRadius;
        char file1[ 50] ;
        char file2[ 50] ;
        pfuWidget *HideWidget;
        int HideToken;

        // Buffer for input device data
        fastrakBuffer *fsData;
        birdBuffer *bdData;
    };

#endif
```

In addition to the new definitions for the pure virtual functions from the abstract base class, other member functions and data present are application specific. Note that some routines in the class make application data available to external routines. This is to facilitate the operation of the general event handler functions.

In particular, the `HideWidget` and `HideToken` variables are used by this application to add a button to the System GUI. Also notice the two input device data buffer variables. Inspection of the application code and the following discussion should reveal how applications can read data from multiple input devices.

The code for defining the class for the `"Space Ship"` application, whose header file is shown above, will now be considered in small parts:

```
// Space Ship Application code implementation file
// Kapil Dandekar

#include "spaceShipapp.h"
#include "hbSysManager.h"
```

```
#include "hbParse.h"

extern hbSystemManager *hbSysMan;

const int window1Pipe = 0;
char *window1Name = "Space Ship Application";
const int window1originX = 400;
const int window1originY = 400;
const int window1sizeX = 400;
const int window1sizeY = 400;

const int secondsActive = 100;
```

This is the start of the file in which the code is defined. Note that the global System Manager is declared as an external variable. Applications seeking to query public System Manager parameters or set System Manager flags need to have this variable declared. Note the use of constants to set some parameters in the particular application. There are three ways that application programmers may store this kind of data:

- class variable,
- predefined constant, and
- data loaded from application data file.

Application programmers may choose any of these options shown above. In our example, we have used the second option purely for instructive purposes. In general, the first or third options are highly recommended.

The constructors and destructor of this class are defined like this:

```
SpaceShipApp::SpaceShipApp(  )
{
    SpaceShipApp( "Space Ship Application", "spaceship.hb" );
        // Invoke application with default name and data file
}

SpaceShipApp::SpaceShipApp( char *newName, char *dataFile ) :
            hbApplication( newName, dataFile )
{
  hbAppWindow *win1Ptr;

  fsData = NULL;
  bdData = NULL;
      // Initialize shared memory area for input devices

  HideToken = 0;
      // Initialize widget token to zero before widget allocation

  allocateWindow( window1Name );
```

```
    allocateWindow( "Click inside me too" );
        // Allocate the two windows of the application

    win1Ptr = getWindow( window1Name );
    win1Ptr->allocateView( "USS Enterprise" );
    win1Ptr->allocateView( "Klingon Ship" );
        // Allocate the two views inside one of the windows

    allocateDevice( POLHEMUS, win1Ptr, sizeof(fastrakBuffer) );
        // Allocate Polhemus device
    allocateDevice( BIRD, win1Ptr, sizeof(birdBuffer) );
        // Allocate Bird device

    win1Ptr->removeEventHandler( ButtonPress );
        // Remove the default button handler for the
        // window (a user-define window specific handler can be
        // added if desired)

    removeEventHandler( ButtonPress );
    addEventHandler( ButtonPress, spinButtonHandler );
        // Remove the default button handler for the application and
        // add a user-defined application specific handler

    removeEventHandler( ClientMessage );
    addEventHandler( ClientMessage, shipClientMessageHandler );
        // Remove the default client message handler for the
        // application and add a user-defined, application specific handler
}


SpaceShipApp::~SpaceShipApp( )
{
    pfuDisableWidget( HideWidget );
}
```

Note that, as mentioned previously, allocation of application resources (windows, views, devices, etc.) occurs in the constructor of the application class. This has the effect of allocating shared storage for all of the resources so that, once multiple processes have been created by the System Manager, information can be shared between the processes. The actual initialization of the objects does not occur until these multiple processes exist; this is the purpose for the addition of the startUp member function in the application.

Event handlers, on the other hand, can be fully specified in the constructor of the application. Both applications and windows have event handlers. Note that in the above example, we remove the default button-press event handler for the first window. Next, the default button-press event handler for the application is removed and replaced with a routine called spinButtonHandler. The implication of making this an application event handler instead of a window event handler is that button-press events received in any window in the application will use the event handler. This eliminates the need to have an identical event handler for each window. Note too, that the second window in the application did not have its button-press handler removed. This means that the default handler, a routine that prints out mouse

event information, remains in effect for the second window. However, the consideration of the "propagation" of application event handlers to window event handlers should be remembered.

The code for these event handlers is shown below:

```
void spinButtonHandler( hbEventHandler &event, void *data )
// User defined button handler
{
  XEvent currentEvent;
  hbAppWindow *winPtr;
  SpaceShipApp *appPtr;

  // Application specific event handling
  currentEvent = event.getXEvent();
  appPtr = (SpaceShipApp *)data;

  switch(currentEvent.xbutton.button)
  {
      case 1: appPtr->setKlgExtentRadius(appPtr->getKlgExtentRadius()/2.0
);
              break;
      case 2: break;
      case 3: appPtr->setKlgExtentRadius(appPtr->getKlgExtentRadius()*2.0
);
              break;
  }

  // Pass the event along to the individual window in which it occurred
  //   (unnecessary, but sometimes useful)
  winPtr = appPtr->getWindow( currentEvent.xany.window );
  if( winPtr )
     winPtr->handleEvent( currentEvent );
}

void shipClientMessageHandler( hbEventHandler &event, void *data )
// User defined client message handler
{
  XEvent currentEvent;
  SpaceShipApp *appPtr;
  hbAppWindow *winPtr;
  int widgetID;
  int hideTokenID;
  pfuWidget *hideWidget;

  // Application specific event handling
  currentEvent = event.getXEvent();
  appPtr = (SpaceShipApp *)data;
  winPtr = appPtr->getWindow( window1Name );

  widgetID = currentEvent.xclient.data.l[0];
```

```
    hideTokenID = appPtr->getHideToken( );
    hideWidget = appPtr->getHideWidget( );

    if( widgetID == hideTokenID )
    {
        if( winPtr )
        {
            if( winPtr->isShown( ) )
            {
                winPtr->hideWindow( );          // Hide the window after first
press
                pfuWidgetLabel(hideWidget, "Kill Task" );
            }
            else
                appPtr->setExitFlag( );         // Kill application after second
press
        }
    }
}
```

Note that in the button handler, events propagate from application to window. In the client message handler, events do not propagate to the windows. Note that both of these event handler functions are not member functions of the Application. Thus, they do not have access to private data or member functions. Public member functions exist for this class, so that event handlers can set and query selected application data.

Remember that the System GUI "communicates" widget selection with applications through the use of client message events. Thus, application programmers seeking to control widgets on the System GUI must have a client-message event handler defined for the widget to have any functionality. In the case above, the application has a single button that hides the window in which the two spaceships are contained after the first button press. At this point, the button changes into a button that kills the application when activated.

The `parseFile` member function is shown below. Note that in this application, the specified application data file contains two file names corresponding to the files containing the model data of the spaceships that will be loaded.

```
void SpaceShipApp::parseFile( )
    // Parse the datafile associated with any particular invocation of
    // the application.
    // The application programmer may decide on the data file format
    // and parsing method.
{
    // Open file
    FILE *fp;
    if( (fp=fopen(getFile(), "r")) == NULL )
    {
        cout << "Cannot open " << getFile() << "!" << endl;
        exit( -1 );
    }
```

```
// Dynamically link Dynamic Shared Objects of specified model files
  hbreadWord( file1, fp );
  cout << "Loader for " << file1;
  if( pfdInitConverter(file1) )
     cout << " dynamically linked" << endl;
  else
     cout << " not found" << endl;

  hbreadWord( file2, fp );
    cout << "Loader for " << file2;
  if( pfdInitConverter(file2) )
     cout << " dynamically linked" << endl;
  else
     cout << " not found" << endl;
  fclose( fp );
}
```

This routine reads the two model file names in question and dynamically links the loader needed to read the model. Again, the file names could be hard coded into the application (not recommended) but, for illustrative purposes, we use the application configuration file.

The next member function we consider is the `startUp` function:

```
void SpaceShipApp::startUp( )
   // Start up the application (invoked by the
   // Application Registry)
{
  hbAppWindow *win1Ptr;
  hbAppWindow *win2Ptr;
  hbAppWinView *viewPtr;
  pfChannel *pfchanPtr;
  hbSysGUI *AppGUI;

  // Initialize the window in the program
  initWindow( window1Name, window1Pipe, window1originX, window1originY,
              window1sizeX, window1sizeY );
  win1Ptr = getWindow( window1Name );
        // Retrieve the window from the Application
        //   window list

  win1Ptr->selectEvents(  KeyPressMask|ButtonPressMask );
                     // Set the X-event mask for the Window

  // Initialize the Views in the Window

  // Enterprise View
  win1Ptr->initView( "USS Enterprise", file1, 0.5, 1.0, 0.5, 1.0 );
  viewPtr = win1Ptr->getView("USS Enterprise");
              // Retrieve the View from the Window View list
```

```
    pfchanPtr = viewPtr->getChannel( );
                    // Retrieve the Performer pfChannel of the View


    Ent_extentRadius = viewPtr->getExtentRadius( );
    pfchanPtr->setChanData( (void *)this, sizeof(SpaceShipApp) );
            // Set the data that will be passed to the View callback
functions
    viewPtr->setAppFunc( spinEntAppFunc );
            // Set a callback function


    // Klingon Ship View
    win1Ptr->initView( "Klingon Ship", file2, 0.0, 0.5, 0.0, 0.5 );
    viewPtr = win1Ptr->getView("Klingon Ship" );
                    // Retrieve the View from the Window View list


    Klg_extentRadius = viewPtr->getExtentRadius( );
    pfchanPtr = viewPtr->getChannel( );
                    // Retrieve the Performer pfChannel of the View
    pfchanPtr->setChanData( (void *)this, sizeof(SpaceShipApp) );
            // Set the data that will be passed to the View callback
functions
    viewPtr->setAppFunc( spinKlgAppFunc );
            // Set a callback function


    // Initialize the second window
    initWindow( "Click inside me too", 0, 400, 0, 200, 50 );
    win2Ptr = getWindow( "Click inside me too" );
    win2Ptr->selectEvents( ButtonPressMask );


    // Initialize the widget
    AppGUI = hbSysMan->getAppGUI( );
    HideWidget = AppGUI->allocateWidget( PFUGUI_BUTTON, win1Ptr, &HideToken
);
    AppGUI->setWidgetY( AppGUI->getWidgetY() - PFUGUI_BUTTON_YINC );
    pfuWidgetDim( HideWidget, AppGUI->getWidgetX( ), AppGUI->getWidgetY( ),
                    PFUGUI_BUTTON_VLONG_XSIZE,
                    PFUGUI_BUTTON_YSIZE );
    pfuWidgetLabel(HideWidget, "Hide Ships" );
}
```

In the first section of this member function, windows (referred to by name) are initialized with a screen origin location and dimensions. Next, the two views inside the first window are initialized. There are several ways to initialize the views and the windows. In this example we use one of the simpler approaches. That is, through specification of a ship graphics model file name, a view is set up of that model with default viewing conditions. To use the full power of Performer, pfChannel can be "manually" created in this part of the code and then added as a view to the System with whatever viewing transformations that are desired.

Also note in the view initialization code, that the views can be referred to by name and have their respective `pfChannel` extracted. This is to allow the application programmer to work within the Hobbes system framework while still having the full power of Performer available. Another very important command to note is the setting of the Performer `pfChannel` data. This command ensures the proper operation of the callback routines. By adding a "`(void *)this`", a copy of the Application object is passed along to all the view callback routines. This information will be available to the callback function, which is not a member function of the application.

Each view has three callback functions: `App`, `Cull`, and `Draw`, which specify any view-specific application processing, cull procedure, or draw routine, respectively. Note that in our example, we only specify a view-specific application processing callback. If there is any application processing that involves anything other than a View, it must be handled in the preFrame or postFrame member function in the application. The "pre" and "post" mean that these routines are invoked just prior to and after, respectively, the initiation of the frame draw and cull processing (`pfFrame`).

After initializing the second window in the application, the System GUI is obtained and appended to by the application. In this specific example, we add a single button to the System GUI. Note the interaction between the widget dimensions and placement with the "current" X and Y location in the GUI itself. Hopefully, future releases of Hobbes will make this widget placement automatic (chosen for the application programmer by the GUI) or better automated (program to add widgets to GUI).

The next section of the code is the definition of the exit condition function, `exitCondFunc` and the exit function `exitFunc`. These two routines are defined like this:

```
int SpaceShipApp::exitCondFunc( )
    // Condition through which application will terminate execution
{
    float time = hbSysMan->getSystemManagerTime( );
    if( (time>secondsActive) )
    {
        cout << "Application egg-timer has run-out" << endl;
        cout << "Application exiting" << endl;
        return( 1 );
    }
    return( 0 );
}


void SpaceShipApp::exitFunc( )
  // Code to execute upon application exit
{
    // Comment/Uncomment the following line to indicate application
    // exit also signifies overall program termination
    //    hbSysMan->setExitFlag( );

    delete this;
        // Call destructor
}
```

The `exitCondFunc` defined for this particular application will cause the application to terminate execution after the global System Manager time reaches a given application defined value. Upon exit, the

`exitFunc` simply invokes the destructor for the object. Currently commented out in the `exitFunc` is the command that would be used to terminate System execution upon application exit.

We now consider the `preFrame` and `postFrame` member functions mentioned earlier in this document:

```
void SpaceShipApp::preFrame( )
    // Application specific, pre frame rendering routine
{
    pfChannel *pChan1;
    pfChannel *pChan2;

    if( !fsData )
    {
        if( deviceExists(POLHEMUS) )
            fsData = (fastrakBuffer *)getData( POLHEMUS );
    }
    else
    {
        fsData->mutex.wait(0);
    /* Uncomment to see Polhemus Fastrak Sensor 1 Data  */

        cout << getName( ) << " Fastrak:";
        cout << "X " << fsData->data.X[ 0] << " ";
        cout << "Y " << fsData->data.Y[ 0] << " ";
        cout << "Z " << fsData->data.Z[ 0] << " ";
        cout << "Az " << fsData->data.Az[ 0] << " ";
        cout << "El " << fsData->data.El[ 0] << " ";
        cout << "Roll " << fsData->data.Roll[ 0] << endl;

        fsData->mutex.signal(0);
    }


    if( !bdData )
    {
        if( deviceExists(BIRD) )
            bdData = (birdBuffer *)getData( BIRD );
    }
    else
    {
        bdData->mutex.wait(0);
    /* Uncomment to see Bird Data */

        cout << getName( ) << " Bird:" ;
        cout << "X " << bdData->data.pos.x << " ";
        cout << "Y " << bdData->data.pos.y << " ";
        cout << "Z " << bdData->data.pos.z << endl;
        bdData->mutex.signal(0);
    }
```

```
    pChan1 = (getView( "USS Enterprise" ))->getChannel( );
    pChan2 = (getView( "Klingon Ship" ))->getChannel( );
    pChan1->passChanData( );
    pChan2->passChanData( );
}


void SpaceShipApp::postFrame( )
    // Application specific, post frame rendering routine
{


}
```

In the preFrame routine, input device data are read and displayed, assuming that the device exists. If the input device does not exist or the Communications Manager is not active, nothing will be displayed. Notice the use of the mutex used to provide mutual exclusion when reading from either input device. Since the data buffer in question is being written to by a different process and can be read by an indefinite number of other applications, this kind of access control is necessary.

In the last part of the preFrame routine, the actual pfChannels of the two Views in the first window are retrieved. These pfChannels are used to pass channel data, at that point of program execution, to whatever View callback functions are defined. It is also very important to know that all time critical commands should be performed in preFrame.